OBSTACLE DETECTION WITH KINECT V2 ON A GROUND ROBOT

By Laurin Fisher

A Project Submitted to the Faculty of the University of Alaska Fairbanks

In Partial Fulfillment of the Requirements

for the Degree of

Master of Science in Computer Science

University of Alaska Fairbanks

December 2018

APPROVED:

Orion Lawlor, Committee Chair
Chris Hartman, Committee Member
Jon Genetti, Committee Member
Chris Hartman, Department Chair
Computer Science Department

Abstract

This paper is about determining whether using a Kinect V2 (Xbox One Kinect) mounted on a LAYLA ground robot can be used to detect obstacles, by generating a heightmap with the depth data. We take several factors into consideration including: framerate, power consumption, field of view, and data noise.

Table of Contents

Introduction

Computer vision has application in search and rescue, disaster response, warehouse stocking, robotic mining, data collection, human recreation and many other fields. Programming and equipping computers and robots to extract useful visual data allows them to perform tasks that are dangerous, mundane, or otherwise expensive for people.

There are many approaches to achieving accurate computer vision, with different sensors and techniques used to detect and classify objects.

In this paper we will discuss our approach to using a Kinect V2 to detect obstacles for a LAYLA robot.

Prior Work

In this paper we chose to use the Kinect V2 sensor, however the original Kinect sensor for the Xbox 360 has been used in several research papers to detect and classify objects, as well as track movement.

Li, Putchakayala, and Wilson in (Li, 2011), sought to detect objects using a Kinect and machine learning. The authors started by placing points into an adjacency matrix and sorting which points were segmented into which objects based on a set distance, color, and angle threshold, enabling objects in 3D space to be formed and labeled. Allowing for 11 office objects to be classified, the authors labeled the objects manually for the algorithm to identify. Training for the algorithm was done on pre-existing scenes with the labels given, then testing without the labels. Their algorithm had an accuracy of 52.11%. With a Kinect mounted on a telepresence robot, the robot took snapshots with the Kinect, turning a full 360 degrees in order to find the desired object in the room. After all the snapshots had been taken and analyzed, the snapshot with the highest matching segment score was chosen and the robot drove towards that part of the

room.

Similar to the experiment in (Li, 2011), Hernández-López et al. in *Detecting objects using color and depth segmentation with Kinect sensor,* produced an algorithm that could target a particular object in a frame using a Kinect's color and depth data. First they started segmenting by color, then the results of that were segmented by a distance threshold. A statistical analysis was then performed to determine which points were in the object without discontinuities. The algorithm was recorded to have a speed of 15 Hz in practice.

Shaun Bond utilized a Kinect V1 and an Oculus Rift to show real world 3D points in a VR headset of a scene in real time (Bond, 2015).

Brian Paden compared consumer sensors, a LIDAR and a Kinect V1, for robot navigation using heightmaps (Paden, 2013). A heightmap involves converting a 3D point cloud into a grid system similar to a 3D bar graph, where the blocks represent the obstacles (or lack thereof) and the highest point in the block is chosen to draw the rectangle.

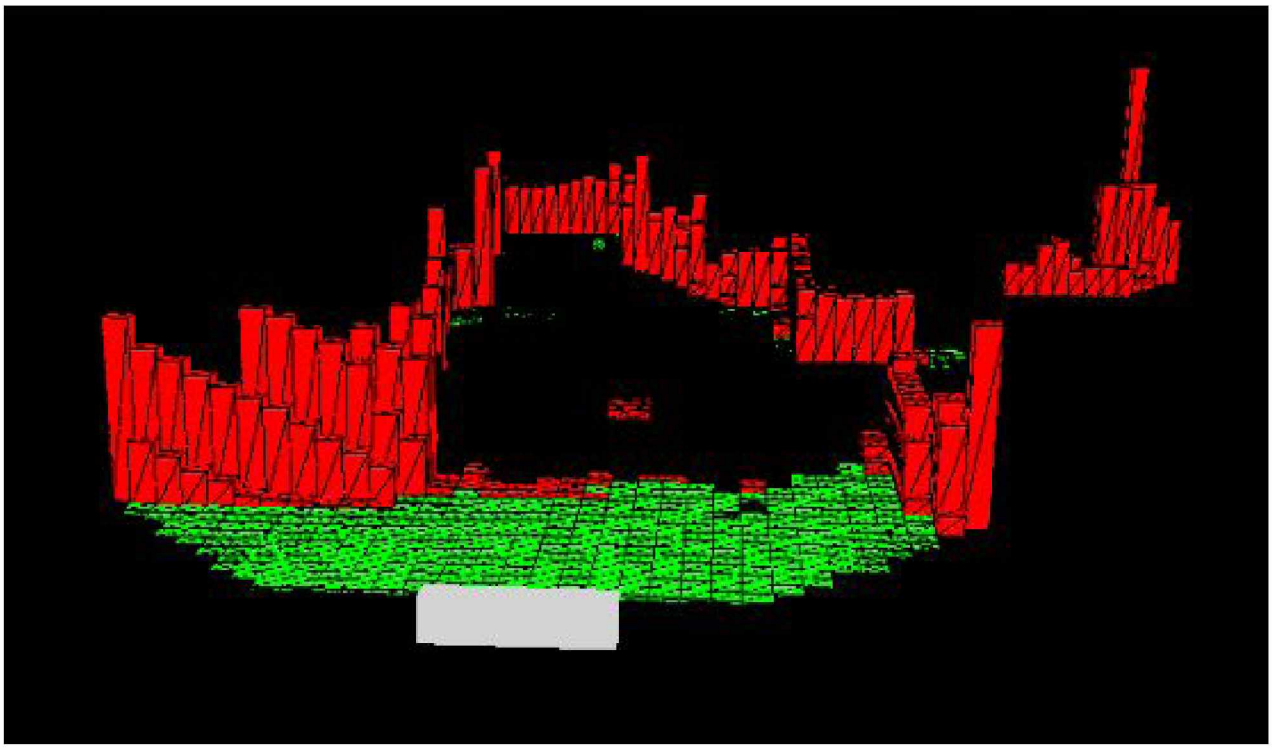


Figure: Hallway Heightmap

While these projects used the Kinect V1 sensor, we chose to use the the most recent Kinect sensor, the Kinect V2, because of its greater field of view and its use of the relatively new time of flight technology.

Sensor Information

Time of flight (ToF) "is based on measuring the total time required for a light signal to reach an object, be reflected by the object, and subsequently be detected by a TOF pixel array," (Bamji, 2015). According to this source, there are two categories for optical ToF methods: stopwatch and accumulated charges principle. The Kinect V2 uses the accumulated charges principle method, which measures the phase difference from the light wave emitted and reflected, to calculate distance (Bamji, 2015).

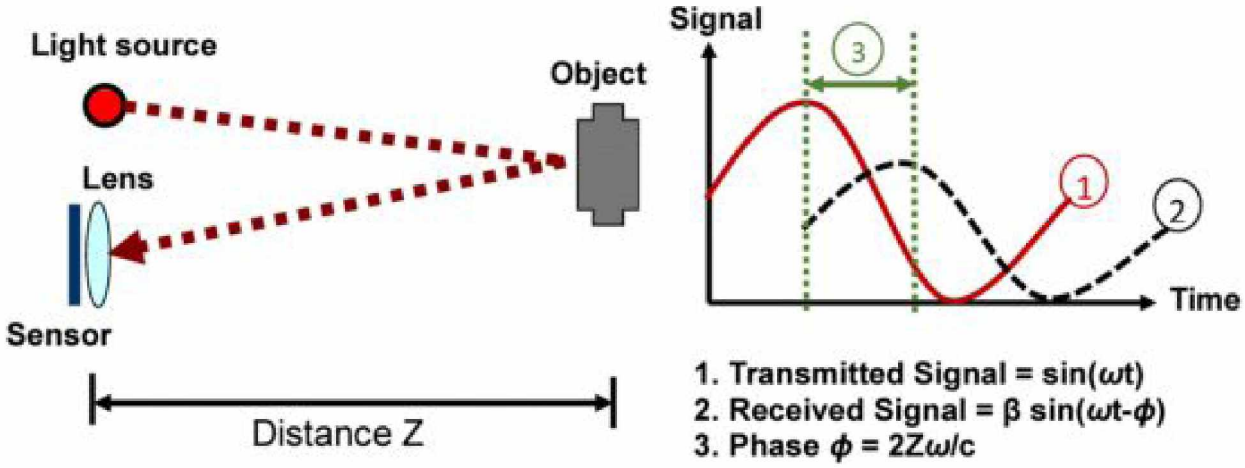


Figure: "Operation principle of tof system" (Bamji, 2015)

This method of depth detection is significantly different from the Kinect V1's infrared dot pattern, and has several big advantages. Whereas parallax (infrared dot pattern) needs two sensor pixels to see a pattern to match, ToF gives a depth for every one sensor pixel. Also, ToF

works in real-time with less dependence on lighting and texturing of objects. By using the newer Kinect, we hoped to improve our range and accuracy through the bigger fov and retrieved pixel images.

Kinect Comparison Table

Device	**Kinect V1**	**Kinect V2**
Technology	Parallax	ToF
Color Camera	640 x 480 @ 30 fps	1920 x 1080 @ 30 fps
Depth Camera	320 x 240 up to 1280 x 1024	512 x 424
Max Depth Distance	~4.5 m, possibly up to 8 m	~4.5 m, possibly up to 8 m
Min Depth Distance	40 cm in near mode	50 cm
Horizontal Depth fov	57 deg	70 deg
Vertical Depth fov	43 deg	60 deg
Tilt Motor	yes	no
USB Standard	2.0	3.0
Supported OS	Win 7 & 8	Win 8
Release Date	2010	2013
Price	$44.99	$99.95

Data from (*Szymczyk, 2014) and the OpenKinect Project*

It is apparent the Kinect V2 is directly better than V1, with an increased fov and increased pixel data count assuming they had the same error rate for data noise.

Although ToF has great advantages (independence of illumination and texture), it also

has its disadvantages. Instead of fast pulses of light based on timing, the Kinect V2 uses continuous wave modulation where continuous light waves are emitted and their phase shift is detected to determine the distance of objects. This opens the Kinect up to disadvantages such as increased noise over time, limited frame rates from computations, multiple reflection, and motion blur. ToF sensors are also susceptible to 'flying pixels' at object boundaries, non-uniformity, and other light sources.

Approach

For our project, we chose to go with height mapping due largely to Paden's success, its computational simplicity, and its usefulness in quick decision making for a robot acting in real time. We chose to mount the sensor on the telepresence robot LAYLA due to LAYLA's pre-existence and room to expand its navigation abilities.

Initially our approach involved gathering 3D points from the Kinect, filtering out noise, and setting the highest point as the point for the specified bin. (We separated out our 3D points into bins based on their X and Z coordinates.) However, we found choosing the highest point was prone to errors, like the Kinect recording the ceiling and thus resulting in areas that were determined undriveable, though the robot could easily drive underneath them. To correct for this, we chose to not look at any points that were higher than what the robot could easily travel underneath. Then we took the trimmed mean of all the points that fell into a particular bin. This ensures that outlying points the Kinect mistakenly recorded and objects like the ceiling or chandeliers would not skew our map.

Implementation

Coordinate Systems

The Kinect's, right-handed, coordinate system consists of Z (depth) directing out of the camera, Y (height) facing upwards, and X pointing to the left of the camera.

Figure: Kinect V2 Coordinate System - Camera Space (Meyer, 2014).

First we convert the 3D points from camera space to world space. Using a left-handed coordinate system, we defined our world space as having positive X to the right, positive Z out from the camera, and positive Y upwards.

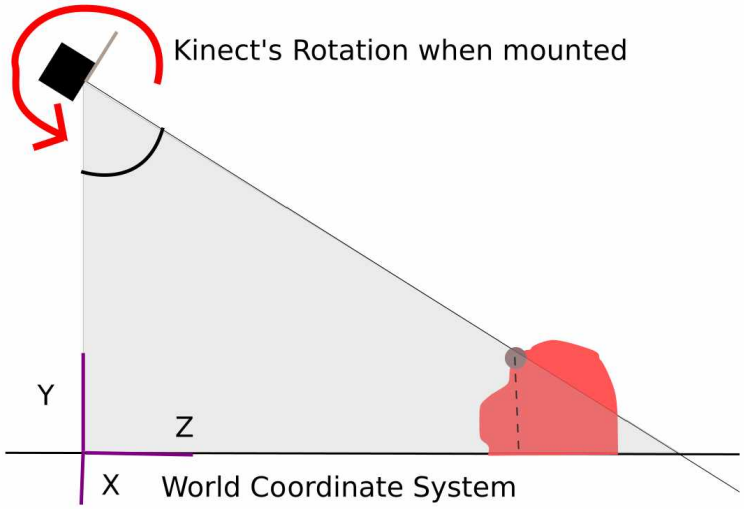


Figure: World Coordinate System

The angle and height of the sensor were static parameters fed into the program, as the sensor was to be secured tightly on the robot.

However, upon inspection and debugging by using an empty room and our visualization output, we discovered our expected coordinate system from the Kinect was incorrect. For this project we were using an open source SDK called libfreenect2 for linux, because Microsoft's SDK was strictly for Windows. We found using this SDK, we had to flip the Y axis as well in order to convert to world coordinates as expected.

In practice, we found this was not our only source of error and decided to do a data analysis of the sensor, collecting error rates, as well as framerates.

Data Analysis

First off, we recorded the frame rate from the sensor both with and without using the viewer, to determine if our algorithm was running in real time. The machine we used for these tests was a Toshiba Satellite P745 with an Intel i5 2.30GHz processor, 6GB of DDR3 RAM, and an Intel integrated graphics 2nd generation core processor.

Frame Rate Table

Run Scenario	Frame Rate (Hz)
Raw data	36
Raw data w/ window	31
Alg w/ window	19
Alg w/o window	31

The results of our frame rate runs suggested our algorithm ran much smoother without the viewer, which could suggest any future visualizations on the local computer could drop the frame rate significantly.

Secondly, we recorded our raw data (before the world coordinate transformation) in static scenarios. We setup scenes where blank walls and the ceiling were in view of the sensor, so we

knew which points were correct. This allowed us to graph our data and determine data noise.

Blank Ceiling

First, we pointed our Kinect at the ceiling and wrote the raw data to a file. We then scatter plotted it using R, and calculated the error of the points in the depth (Z) direction.

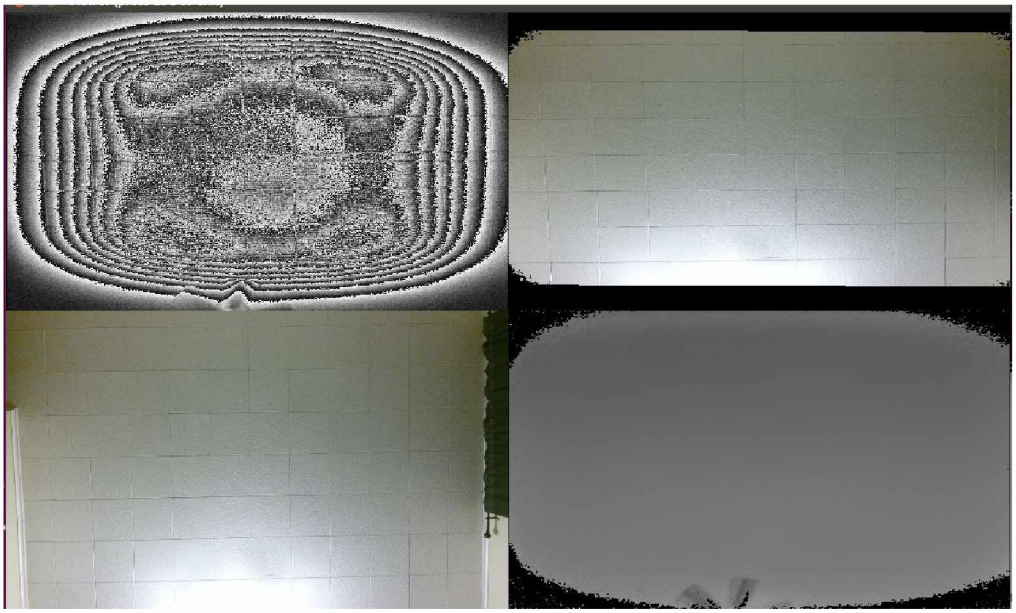


Figure: Viewer output of the ceiling scene

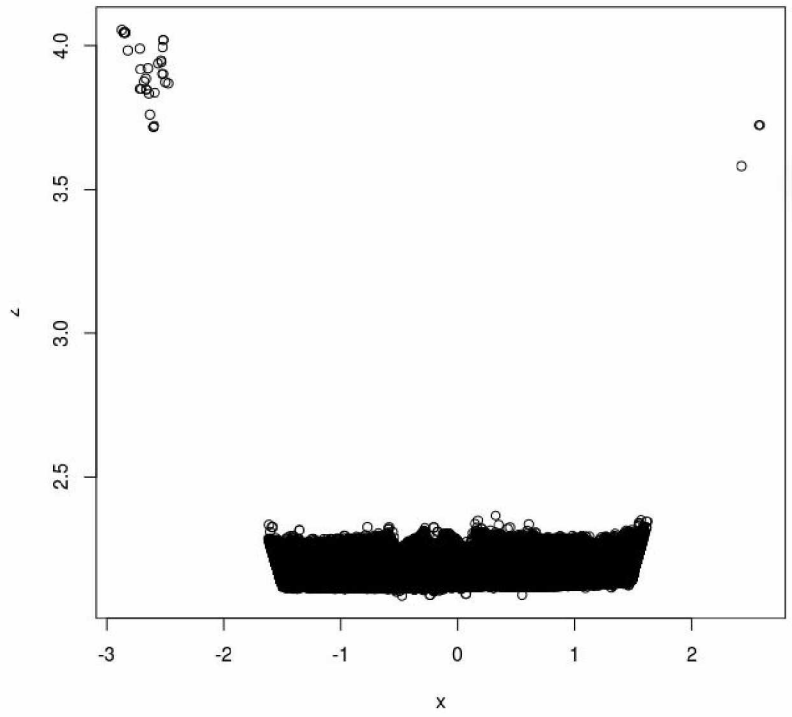


Figure: Scatter plot of X and Z (depth) point data in R

Ceiling Scene Error Rate Table

	Run 1	**Run 2**
# Points that were Errors	34	4
Total Points	206534	206909
% Error	0.016%	0.0019%

We determined that the depth data was incorrect if it was beyond the 2.5 meter mark. Although the ceiling in the scene was not curved, and therefore should have been a consistent depth value (straight line), we gave the points more error room in case the Kinect had been slightly tilted on its stand.

Wall with empty floor

Our second scene involved the wall and the floor. We calculated the error in the Y (height) axis direction. Note this is before the world coordinate transformation of points, therefore Y coordinates are flipped in the graph below (highest is 0.5 m not 0.1 m).

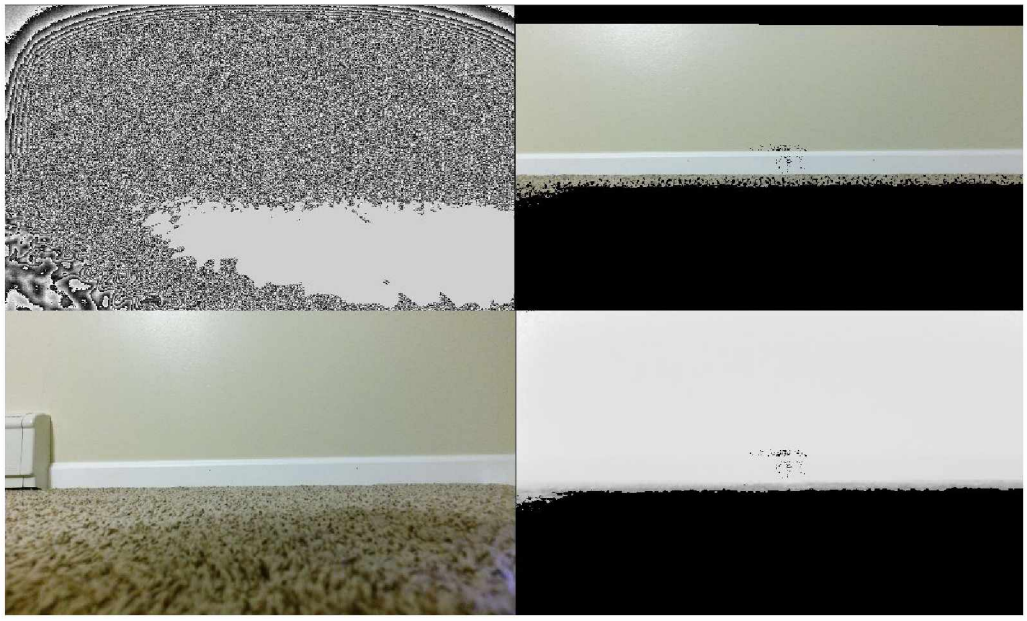


Figure: Viewer output of wall scene

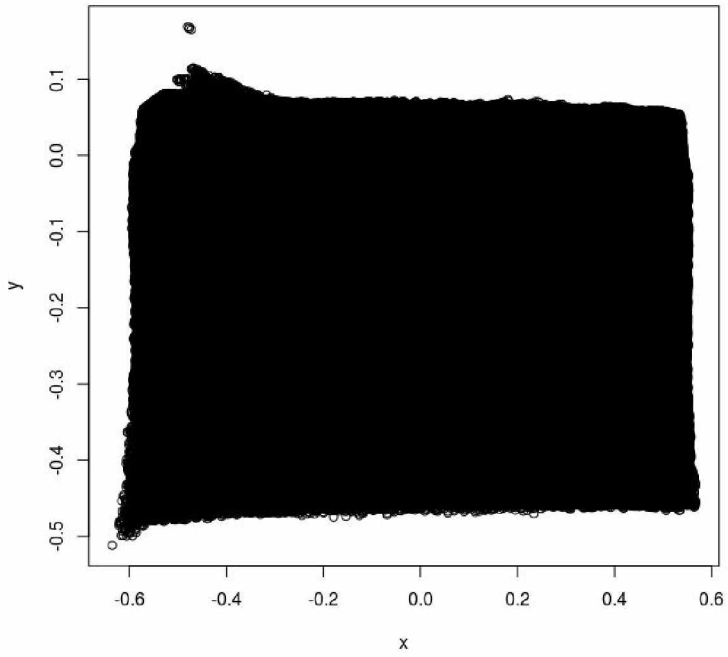


Figure: Scatter plot of X and Y (height) values in R

Wall Scene Error Rate Table

	Run 1	**Run 2**
# Points that were Errors	10361	10499
Total Points	128469	128675
% Error	8%	8%

We found the error rates for Z (depth) were very close to zero, whereas the Y direction (height) was prone to much more error, calculated at around 8%.

Object Detection

We tested how the Kinect could see objects of different sizes at different distances. We

tested a pen, around 1 cm in height, and a binder with a 3 cm height on its largest side.

We determined the Kinect could not see the pen even if it was 60 cm away (just a little beyond the sensor's beginning viewing range). The pen appeared to be too small for the Kinect to pick up, even in the viewer coloration. However in this particular case, LAYLA's 11 inch diameter wheels would be able to drive over such an object.

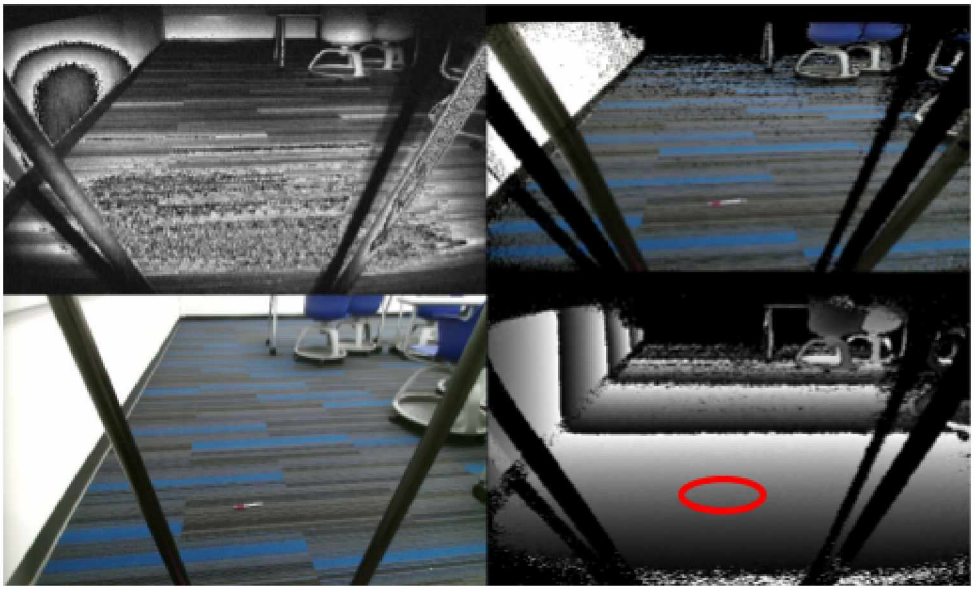


Figure: A pen 60 cm away from the sensor shown in the viewer

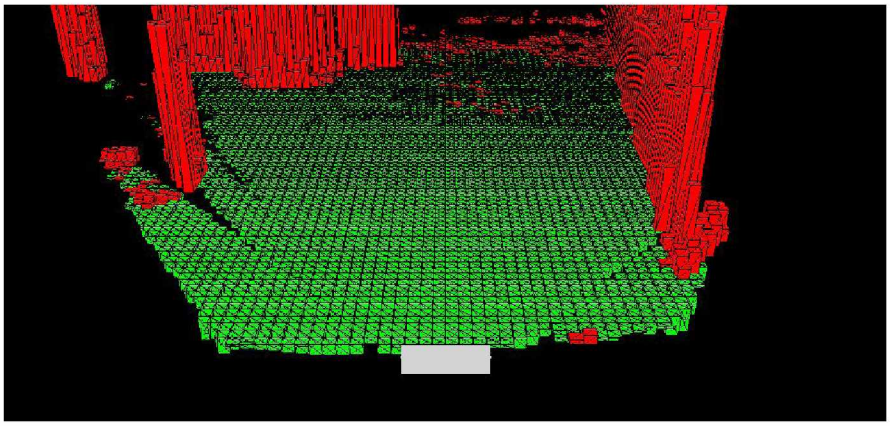


Figure: Visualization for pen scene

For our second item we chose a white binder that had a height of 3 -+ cm on its highest

side facing the sensor.

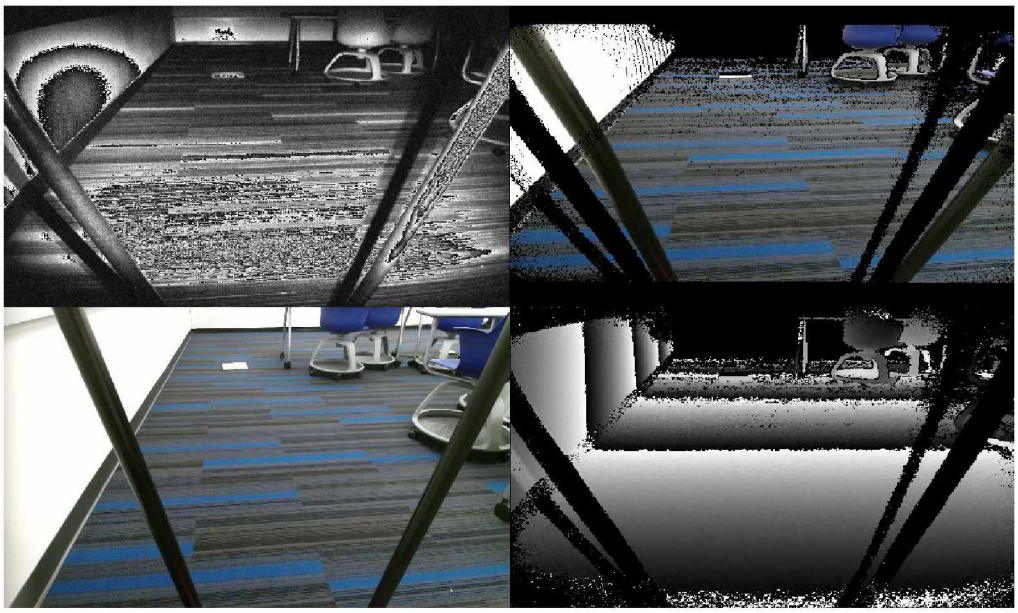


Figure: White binder 3.4 m away

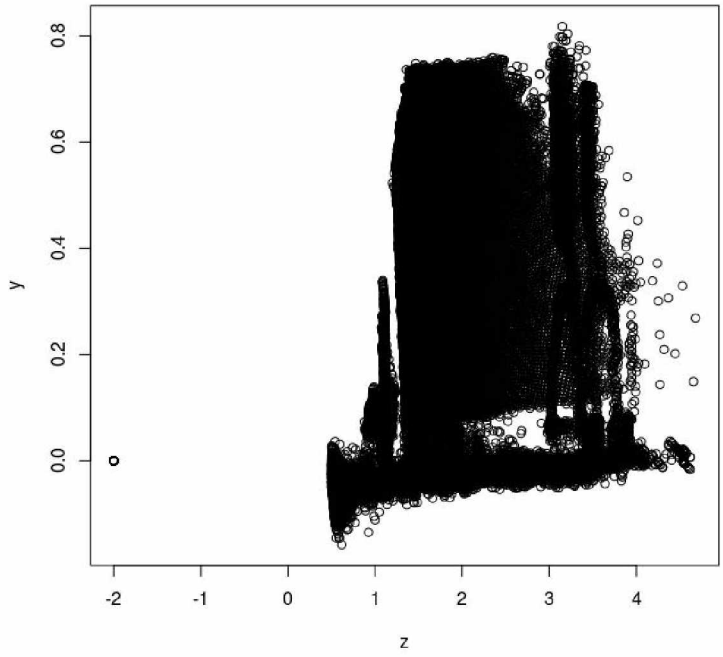
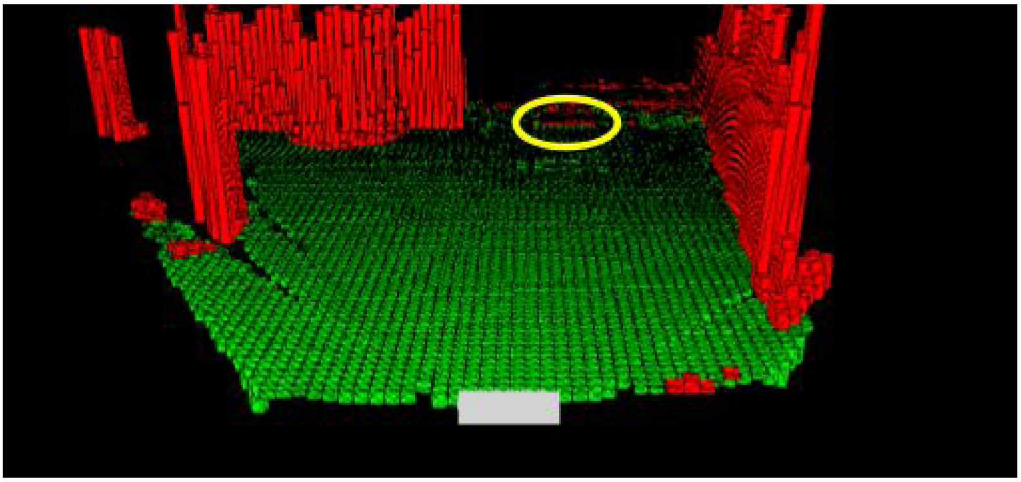


Figure: R graph of Y (height) and Z (depth) data of points

We see that there are points at 3.4 m that are 3 cm or 0.03 m tall. However, from the visualization it was much easier to see the Kinect picked up on the binder.



Binder: 3 cm tall - 3.42 meters away from Kinect

We moved the binder slightly farther to around 4 m away, the very of the edge of the Kinect's proposed fov.

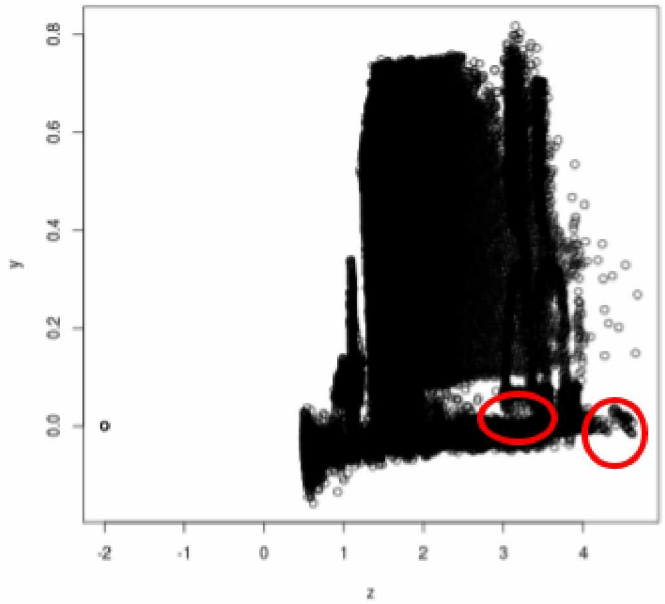


Figure: R graph of Y (height) and Z (depth) data of points

By looking at the R graph, the binder is no longer at 3.4 meters, and some additional points appear around 4 m.

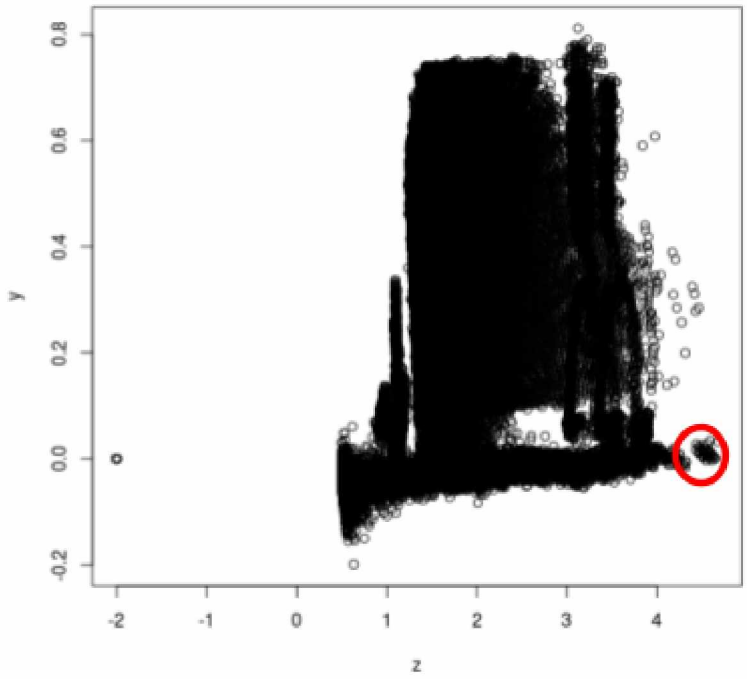


Figure: R graph of Y (height) and Z (depth) data of points

The points themselves suggest the Kinect can see the binder still, though no data is collected any further, which is proven by the Kinect not seeing more floor further away.

Figure: Viewer output of binder 4 m away

Although the points suggest the sensor can see the binder, looking at the visualization it is indistinct, and we could barely see the dark grey the viewer output showed to color the object.

Power Consumption

The Kinect's AC adapter claimed to require 12 volts at 2.67 amps from LAYLA's battery to function. Although when tested, the Kinect was able to use around 1.5 amps and 11 volts.

Conclusion

In the end, LAYLA bore the Kinect's weight, battery, and computational power needs. The Kinect in return was able to produce obstacle detection data with a raw data error rate of 8%. With our added filtering and trimmed mean calculations, the obstacles we placed in the scenes were detected accurately (unless they were pencil sized). Testing in a standard home environment, with hardwood, lightly colored walls, and carpet, the Kinect performed extremely well. In this environment, this approach would be enough for LAYLA (or any other robot that

could navigate over pencil sized objects) to navigate indoors safely.

Future Work

Although we concluded the Kinect V2 could detect obstacles to aid in floor robot navigation, and LAYLA could handle the sensor's power needs, we did not test how the Kinect would handle the outdoors, other light-emitting sensors, or reflective surfaces. Such research would be very applicable to other environments, such as a building with clean, reflective floors, avoiding other robots with light-emitting sensors, or navigating outside on sidewalks. We welcome contributions to our library available at https://github.com/Dev-Laurin/MastersProject.

Appendix

Protonect.cpp (Main Loop):

```
//////////////////////Height Map Code//////////////////////////////
      registration->apply(rgb, depth, &undistorted, &registered,
true);

      //MARK: Getting Point Data
      vector<Point>
framePoints(undistorted.width*undistorted.height);
      int index = 0;
      for(unsigned int i=0; i<undistorted.height; ++i){
        for(unsigned int j=0; j<undistorted.width; ++j){
          float x,y,z;

          //IF color is not needed can use getPointXYZ() for faster
            //computation
          registration->getPointXYZ(&undistorted, i, j, x, y, z);
          Point p(x,y,z);
          framePoints[index] = p;
```

```
            ++index;
          }
        }

        //Transform Points to World Space
        transformPoints(framePoints, xAxisAngleRotation, cameraHeight);

        //Filter bad points
        filterPoints(framePoints, heightClearance);

        //MARK: Segment Into Bins
        //place points into bins based on their x & z value
        segmentIntoObjects(framePoints, binSize,
          maximums, bins, kinectMinX, kinectMinZ, xAxisAngleRotation,
          cameraHeight, heightClearance);

      //Save all points gathered in bins to file for statistical
analysis

        //MARK: Save maximums to JS file for viewing later
      ofstream jsFile("data.js");
      initJSFile(jsFile, "grid");
      writeToJS(maximums, jsFile);

      endWritingPoints(jsFile);
      writeVariable(jsFile, width, width);

      jsFile.close();

////////////////////////////////////////////////////////////////////////
```

Transform Points to World Coordinates:

```
void transformPoint(Point &point, double &xAxisRotationAngle,
      double &cameraHeightFromGround){

      //X axis reversed
```

```
	point.x = -point.x;

	point.y = -point.y;

	//multiply by the rotation matrix
	if(xAxisRotationAngle != 0){
		//save for math correctness
		Point kinect = point;
		//Y
		point.y = (kinect.y * cos(xAxisRotationAngle))
			- (kinect.z * sin(xAxisRotationAngle));
		//Z
		point.z = kinect.y * sin(xAxisRotationAngle)
			+ kinect.z * cos(xAxisRotationAngle);
	}

	//add translation transform to point
	point.y = point.y + cameraHeightFromGround;
}

void transformPoints(vector<Point>&points, double&
xAxisRotationAngle, double& cameraHeightFromGround){

	for(unsigned int i=0; i<points.size(); i++){
		//Filter out nan values
		if(std::isnan(points[i].x) || std::isnan(points[i].y) ||
			std::isnan(points[i].z)){

			//An incorrect z = point is invalid (kinect cant see
behind it)
			points[i] = Point(0.0,0.0, -2.0);
		}

		transformPoint(points[i], xAxisRotationAngle,
cameraHeightFromGround);
	}
}
```

Filtering out Points:

```
//Filter noise and bad data out
void filterPoints(vector<Point>&points, double & heightClearance){

      for(size_t i=0; i<points.size(); i++){

            //if the point is higher than the robot
            if (points[i].y> heightClearance){
                  points[i] = Point(0.0, 0.0, -2.0);
                  continue; // robot won't hit this
            }

          //Check if this depth is plausible--agrees with neighbors
          double noise=0.1; //

          //Left neighbors
          for (int nbor=-2;nbor<0;nbor++)
            if (distanceSquared(points[i],points[i-nbor])>noise*noise)
                  points[i] = Point(0.0, 0.0, -2.0);

          //Right neighbors
          for(int nbor=1; nbor<3;nbor++)
            if (distanceSquared(points[i],points[i+nbor])>noise*noise)
                  points[i] = Point(0.0, 0.0, -2.0);
      }
}
```

Segment Points into Bins:

```
//Filter, transform, and segment the 3D points into a 2D vector
void segmentIntoObjects(vector<Point>& threeD,
      double &binSize, vector<vector<Point>>&maximums,
      vector<vector<binData>>&bins, double & kinectMinX,
```

```
	double & kinectMinZ, double & xAxisRotationAngle,
	double & cameraHeightFromGround, double &heightClearance){

	for(unsigned int i=0; i<threeD.size(); i++){

		//If point has a negative Z it is invalid. The kinect
		//cannot see behind itself.
		if(threeD[i].z < 0){
			continue; //next loop iteration
		}

		//Place into 2D vector bins
		double x = threeD[i].x;

		//Round the value up to next int
		int index = (x - kinectMinX)/binSize;

		//place point into bins
		int z = (threeD[i].z - kinectMinZ)/binSize;

		//add point to bin
		bins[index][z].data.push_back(threeD[i]);

		//Bins Mean Trimmed Variables
		//SUM
		bins[index][z].sum += threeD[i].y;
		//Maximum
		if( bins[index][z].maximum < threeD[i].y)
			bins[index][z].maximum = threeD[i].y;
		//Minimum
		if( bins[index][z].minimum > threeD[i].y )
			bins[index][z].minimum = threeD[i].y;
		//Total points
		bins[index][z].totalPoints++;
		//Squared Sum
		bins[index][z].squaredSum += bins[index][z].sum *
bins[index][z].sum;
		//Variance so far
```

```
            /*
          //Get Variance without storing all data until end and
looping through
          First, I added up all of the numbers:1 + 2 + 3 + 4 + 5 = 15
          I squared the total, and then divided the number of items
in the data set 15 x 15 = 225
          225 / 5 = 45
          I took my set of original numbers from step 1, squared them
individually this time, and added them all up:(1 x 1) + (2 x 2) + (3
x 3) + (4 x 4) + (5 x 5) = 55
          I subtracted the amount in step 2 from the amount in step
3:55 - 45 = 10
          I subtracted 1 from the number of items in my data set:5 -
1 = 4
          I divided the number in step 4 by the number in step 5:10 /
4 = 2.5
          This is my Variance!
          Finally, I took the square root of the number from step 6
(the Variance),
          √(2.5) = 1.5811388300841898
          This is my Standard Deviation!
          ///From:
https://www.statisticshowto.datasciencecentral.com/calculators/varian
ce-and-standard-deviation-calculator/
          */
          bins[index][z].trimmedMean = (bins[index][z].sum -
bins[index][z].maximum - bins[index][z].minimum) /
(bins[index][z].totalPoints - 2);
           double trimmedMean = bins[index][z].trimmedMean;

           threeD[i].y = trimmedMean;
           if(std::isinf(trimmedMean) || std::isnan(trimmedMean))
                threeD[i].y = bins[index][z].maximum;
           maximums[index][z] = threeD[i];
     }

}
```

Program Arguments for Creating Bins:

```
 ////////////////////Height Map Code////////////////////////
 //Can change these as needed (see args first)
  //Variables needed for coordinate change to world coords
  double cameraHeight = 0.0; //meters
  double xAxisAngleRotation = 0; //degrees
  double heightClearance = 5.0; //meters
  double drivableCM = 1.0; //at what centimeter value can the robot
drive over?

  //Variables needed for bin creation/height map
  //The Kinect's width and depth that's viewable by sensor in Meters
 double kinectMinX = -3.2;
 double kinectMinZ = 0.4;
 double binSize = 0.04; //meters
 double kinectMaxXWindow = 6.4; //Meters
  ////////////////////////////////////////////////////////////
```

References

Al-Kaff, A., Moreno, F. M., Escalera, A. D., & Armingol, J. M. (2017, October 30). Intelligent vehicle for search, rescue and transportation purposes. Retrieved from https://ieeexplore.ieee.org/document/8088148/

Bamji, C. S., O'Connor, P., Elkhatib, T., Mehta, S., Thompson, B., Prather, L. A., Snow, D., Akkaya, O., Daniel, A., Payne, A., Perry, T., Fenton, M., Chan, V. (2015, January). A 0.13 µm CMOS System-on-Chip for a 512 × 424 Time-of-Flight Image Sensor With Multi-Frequency Photo-Demodulation up to 130 MHz and 2 GS/s ADC. Retrieved from https://ieeexplore.ieee.org/document/6964815

Bond, S. P. (2015, April). Projecting Physical Objects into a Virtual Space using the Kinect and Oculus Rift. Retrieved from www.cs.uaf.edu/2015/spring/ms/MS_CS_Shaun_Bond.pdf

Castaneda, V., & Navab, N. (2011, January 6). Time-of-Flight and Kinect Imaging. Retrieved from campar.in.tum.de/twiki/pub/Chair/TeachingSs11Kinect/2011-DSensors_LabCourse_Kinect.pdf

Coordinate mapping. (2014, October 20). Retrieved from docs.microsoft.com/en-us/previous-versions/windows/kinect/dn785530(v=ieb.10)

Dalal, N., & Triggs, B. (2005, July 25). Histograms of oriented gradients for human detection. Retrieved from ieeexplore.ieee.org/xpls/icp.jsp?arnumber=1467360

Duan, R., Yue, G., Feng, C., Tian, Y., Yu, S., Foulds, R., & Su, H. (2018, April 9). Real-Time Robust 3D Plane Extraction For Wearable Robot Perception and Control. Retrieved from haosu-robotics.github.io/images/Paper/2018DMD-CameraSensingForWearableRobotControl.pdf

Ess, A., Leibe, B., Schindler, K., & Gool, L. V. (2009, May 12). Moving Obstacle Detection in Highly Dynamic Scenes. Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5152884

Hansch, R., Kaiser, S., & Helwich, O. (2017). Near Real-time Object Detection in RGBD Data. Retrieved from http://www.scitepress.org/Papers/2017/61014/61014.pdf

Henry, P., Krainin, M., Herbst, E., Ren, X., & Fox, D. (2010). Rgbd mapping: Using depth cameras for dense 3d modeling of indoor environments. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.188.7365

Hernandez-Lopez, J., Quintanilla-Olvera, A., Lopez-Ramirez, J., Rangel-Butanda, F., Ibarra-Manzano, M., & Almanza-Ojeda, D. (2012, May 23). Detecting objects using color and depth segmentation with Kinect sensor. Retrieved from http://www.sciencedirect.com/science/article/pii/S2212017312002502

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. Retrieved from https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

Li, T., Putchakayala, P., & Wilson, M. (2011). 3D Object Detection with Kinect. Retrieved from www.cs.cornell.edu/courses/cs4758/2011sp/final_projects/spring_2011/Li_Putchakayala_Wilson.pdf

Madani, K., Fraihat, H., & Sabourin, C. (2017, November 7). Machine-awareness in indoor environment: A pseudo-3D vision-based approach combining multi-resolution visual information. Retrieved from ieeexplore.ieee.org/document/8095116/

Mallick, S. (2016, December 6). Histogram of Oriented Gradients. Retrieved from https://www.learnopencv.com/histogram-of-oriented-gradients/

Mallick, S. (2016, November 14). Image Recognition and Object Detection : Part 1. Retrieved from www.learnopencv.com/image-recognition-and-object-detection-part1/

Obaid, W., Rabie, T., & Baziyad, M. (2017, October 23). Real-Time Color Object Recognition and Navigation for QUARC QBOT2. Retrieved from ieeexplore.ieee.org/document/8079758/

Osman, H. I., Hashim, F. H., Zaki, W. M., & Huddin, A. B. (2017, October 19). Entryway detection algorithm using Kinect's depth camera for UAV application. Retrieved from ieeexplore.ieee.org/document/8070572/

Paden, B. (2013, December). Mobile Robot Path Planning using a Consumer 3D Scanner. Retrieved from www.cs.uaf.edu/media/filer_public/3d/63/3d634ca2-c963-4ae8-8dc5-ca0e5eed6625/mswe_brian_paden.pdf

Pham, T. T., Nguyen, H. T., Lee, S., & Won, C. S. (2017, January 5). Moving object detection with Kinect v2. Retrieved from http://ieeexplore.ieee.org.proxy.library.uaf.edu/document/7804827/

Prajapati, A., Bhatt, M., Joshi, Y., & Negi, R. (2018, March). Skeleton Tracking Using Microsoft Kinect for Windows on Matlab. Retrieved from www.ijrra.net/Vol5issue1/IJRRA-05-01-85.pdf

Schwarz, M., Milan, A., Periyasamy, A. S., & Behnke, S. (2017, June 20). RGB-D object detection and semantic segmentation for autonomous manipulation in clutter. Retrieved from https://journals.sagepub.com/doi/abs/10.1177/0278364917713117?journalCode=ijra

Strbac, M., Markovic, M., & Popovic, D. B. (2013, January 28). Kinect in neurorehabilitation: Computer vision system for real time hand and object detection and distance estimation. Retrieved from ieeexplore.ieee.org/xpls/icp.jsp?arnumber=6419983

Szymczyk, M. (2014, December 9). How Does The Kinect 2 Compare To The Kinect 1? Retrieved from zugara.com/how-does-the-kinect-2-compare-to-the-kinect-1

Viola, P., & Jones, M. (2003, April 15). Rapid object detection using a boosted cascade of simple features. Retrieved from ieeexplore.ieee.org/document/990517/authors